Newton-Raphson Integer Division for Area-Constrained Microcontrollers

Nima Badizadegan
T0 Labs
Email: nima@t0labs.com

***Abstract*—Many small microcontrollers today are equipped with single-cycle multipliers, but use division algorithms that compute only one bit at a time, resulting in division operations with up to 32 (or more) CPU cycles of latency, stalling the core during computing. This is primarily due to area constraints, since fast division algorithms based on iterative approximation require relatively large lookup tables to produce a useful speedup over slower algorithms.**

We propose an alternative to using lookup tables for the initial approximation. Instead, we augment the single-cycle multiplier to compute a heavily-quantized polynomial approximation of 1/D. The resulting approximation has 8-bit precision and computes in a single cycle at a cost of 400 added logic gates.

Finally, we demonstrate a state machine that performs 32-bit integer division using the augmented multiplier and a microcontroller ALU to compute quotient and remainder with 3–11-cycle latency. When synthesized in a MAX 10 FPGA, the datapath with fast division used 3,260 logic elements, compared to 2,759 LEs for a microcontroller datapath without division, an area increase of only 18%, or 4x less than the area of a single-cycle multiplier.

I. Introduction

Many small microcontroller cores operate under tight area and power constraints, which make large arithmetic circuits infeasible to use. However, even when cores are too small for a floating-point unit, many of these cores still offer a full suite of multiplication and integer division instructions, particularly those compliant with the RISC-V M instruction extensions [1]. These microcontrollers often use a single-cycle multiplier, but only a radix-2 division unit [2], [3]. As a result, while multiplication on these cores is fast, division instructions can stall the core for 32 cycles or more.

In contrast, larger CPUs for real-time control and application processing often use either high-radix digit-recurrence algorithms [4] or iterative approximation algorithms to compute division results [5], [6]. These algorithms permit division operations to have latency and reciprocal throughput of less than 10 clock cycles [7]. However, iterative approximation algorithms usually use large lookup tables to produce initial reciprocal estimates [8], which limits the applicability of these methods to area-constrained systems.

Thus, research on division for area-constrained systems has focused on enhancements to restoring and non-restoring division algorithms to improve the speed of these methods, including reducing circuit delay [9], adding fast early termination [10], and enhancing microcontrollers with approximate division instructions that can be used to reduce division latency where appropriate [11].

Finally, most of the literature on division circuits is focused on floating-point division, which is fundamentally different than integer division. Floating-point division produces a single rounded result, while integer division has two results: a quotient which is always rounded toward zero, and a remainder [12]. For many integer division operations, including those used for modular arithmetic and divisibility checks, the remainder is the only operation used [13].

We propose to approach the problem of small, fast division units from the other direction: by packing the circuits needed for fast division algorithms using iterative approximation algorithms into the existing functional units in a microcontroller. Newton-Raphson division is computed in two stages: an initial approximation, usually done using lookup tables, and subsequent iteration rounds that provide quadratic convergence, done with FMA circuits [14]. In this implementation, we perform initial approximation using a quantized polynomial approximation instead of a lookup table, and combine the circuit for this approximation with the multiplier. Refinement rounds are then run using a state machine, implemented in microcode, and an integer FMA unit. To avoid widening the datapath, we add a finalization step to the computation, allowing an inexact result after the end of the Newton-Raphson iterations. This finalization step corrects the remainder first, and then the quotient if needed.

The remainder of the paper is organized as follows. Section II shows a derivation of a quantized polynomial approximation of $1/D$ that we will integrate into the multiplier to replace a lookup table. In Section III, we describe a circuit that combines a single-cycle multiplier with a division estimation circuit. A description of a full state machine for Newton-Raphson division using our combined multiplication and division estimation circuit is provided in Section IV, including early termination. Section V presents an evaluation of area and maximum frequency of the division estimation circuit and the division state machine in an FPGA model. Finally, the main conclusions are presented in Section VI.

II. Area-constrained Polynomial Approximation

Polynomial approximation algorithms typically produce approximations that minimize error [15]. However, for division, we do not need to minimize error, but bound it by a target,

e_t. With a quadratic convergence algorithm, an initial approximation with 8-bit precision converges as quickly to a 32-bit result as an initial approximation with 15-bit precision [14]. Thus, between an optimal polynomial approximation from the Remez or Chebyshev algorithm and a minimally-satisfactory polynomial with error $e < e_t$, we have an error budget to spend on making the approximation easier to compute.

Methods such as bit heaps allow efficient computation of known polynomial approximations [16]. However, we would prefer to use an approximation that is smaller and more non-optimal, but still with error less than e_t, such that the final circuit can re-use the structure of an integer multiplier.

We start from an equation form that reduces the number of required multiplications and can be computed with narrow bit widths. For example, a 6th order polynomial in expanded form takes the equivalent of 5 multiplications to produce powers of x, as well as several full-width additions. However, in a factored form like,

$$P(x) = \xi_0(\xi_1 + \xi_2 x + x^2)(\xi_3 + \xi_4 x + x^2)(\xi_5 + \xi_6 x + x^2)$$

we can compute $P(x)$ using 3 non-constant multiplications. Additionally, constant coefficients of x and the individual quadratic sub-expressions can be heavily quantized. Each polynomial coefficient in the expanded form is the product of multiple ξ_i, so the loss from quantization is less severe than in an expanded form. Additionally, the fixed-point products of the sub-expressions are largely determined only by the high-order bits of each expression, resulting in less quantization loss than an expanded equation form.

A. Finding an Easy-to-Compute Polynomial for $1/D$

A fourth-order Remez approximation for $1/D$ with $D \in (0.5, 1)$ yields

$$R_{1/D}(x) = 5.17028(1.96866 - 2.6834x + x^2) \times (0.703212 - 1.02432x + x^2) \quad (1)$$

To reduce this expression to a computable form, we first quantize coefficients and then quantize sub-expressions while computing optimal constants for each sub-expression. Finding sets of quantized coefficients can be done entirely before quantizing product terms, but quantizations of each k_n are not independent due to the multiplication of the two factors. Candidate coefficient sets for 8-bit division with this form of polynomial can be found in Table I. The best approximation is:

$$A_{1/D}(x) = 5(c_1 - 2.6875x + x^2)(c_2 - 1.03125x + x^2) \quad (2)$$

Given a set of quantized coefficients, we choose widths of each multiplication while setting c_1 and c_2 to minimize error. The terms that can be quantized are the input to the x^2 calculation, the two factors inside the parentheses, and the final multiplication. This can be done by brute force, using minimax for c_1 and c_2 at each trial, or using a SAT or LP solver.

With $A_{1/D}$ from equation 2, the following quantization was selected:

TABLE I
CANDIDATE QUANTIZATIONS OF COEFFICIENTS OF EQUATION 1

Coefficients (Decimal and Binary)			
k_1	k_2	k_3	Accuracy (Bits)
5.17028	-2.6834	-1.02432	11.2
5	-2.6875	-1.03125	10.2
101	-10.1011	-1.00001	
4	-2.6875	-1.03125	8.8
100	-10.1011	-1.00001	
5	-2.75	-1.03125	8.5
101	-10.11	-1.00001	
5	-2.6875	-1	8.2
101	-10.1011	-1	

TABLE II
COMPUTED CONSTANTS FOR EQUATION 2 IN DECIMAL AND FIXED POINT

Constants		Accuracy Bounds	
c_1	c_2	$\min(A_{1/D}(x)x)$	$\max(A_{1/D}(x)x)$
Most Accurate			
1.97968	0.71753	0.99814	1.00178
0x7eb3	0xb7b	0xff85d1cc	0x10074af66
Underestimate			
1.97925	0.71729	0.99664	0.99996
0x7eac	0xb7a	0xff2413c5	0xfffd77a1
Overestimate			
1.97979	0.71777	1.00006	1.00369
0x7eb5	0xb7c	0x10003dbb7	0x100f1b092

- Compute x^2 with 13-bit input width
- Compute $p_1 = (c_1 - 2.6875x + x^2)$ with 14-bit precision
- Compute $p_2 = (c_2 - 1.03125x + x^2)$ with 12-bit precision
- Multiply p_1 and p_2 at full precision (12b $\times$ 14b)

The resulting circuit achieves an accuracy level of slightly more than 8 bits, compared to 10.2 bits unquantized, but is minimally-sized, taking less than 10 narrow additions and 2 small multiplications to fully compute. The constants found for this approximation are shown in Table II, for precise estimation, overestimation, and underestimation.

III. AUGMENTED MULTIPLIER CIRCUIT ARCHITECTURE

Our implementation starts with a 32-bit full multiplier using Booth encoding, and multiplexes the array of adder trees to compute sums and products for the chosen polynomial. Other multiplier topologies, including array multipliers, can also serve as a host for our division approximation. For signed and unsigned multiplication, the Booth code multiplier computes the sum of 17 partial products, which are summed using adder trees to produce a 64-bit product.

The same array of adder trees is used to compute the polynomial for division estimation. The mapping of division approximation terms onto existing multiplier partial products is shown in Figure 1. The two multiplications, computing x^2 and $5p_1p_2$, make use of the Booth encoding circuits for the respective rows that contain their partial products, while the additions that compute p_1 and p_2 bypass the Booth

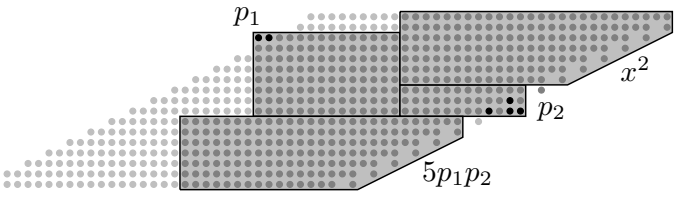

Fig. 1. Dot diagram showing mapping of the division approximation polynomial onto existing multiplier partial products (grey dots), and added nonzero terms (black).

encoders and multiplex directly into the columns of adder trees. The computation of p_1 and p_2 adds 6 nonzero bits to the multiplier array, but all other computations fit entirely inside the multiplier. With an array multiplier, all of the computations would fit.

To facilitate the multiplier array cuts required for the addition of the division approximation, the summation trees used for each column are shown in Figure 2. The trees are not simple Wallace trees: we use a mix of 3:2 and 4:2 compressors to allow the trees to be split at the 7th and 10th partial products to cover each of the distinct computations happening in the array.

The top 7 partial products are used to both compute $(c_1 - 2.6875x)$ and x^2. The low-order bits of x^2 are summed using an adder circuit to produce a carry signal that is passed to the computations of p_1 and p_2. The computation of p_1 is finished using the first multiplexer, which brings x^2 from columns 12–25 over to columns 26–39, the following 4:2 compressor, and an adder. Once p_1 is computed, it feeds into the partial product generation circuits for the bottom seven partial products. Computation of p_2 uses the normal paths through the multiplier, summing x^2 as computed by the top 7 partial products and the linear and constant terms of p_2 in the next 3 partial products. After all 10 components are compressed to two, the final sum is taken before p_2 is passed to the booth encoders for the bottom 7 partial products.

For computation of the final product, $5p_1p_2$, the bottom 7 partial products operate as normal, Booth-encoding p_2 and generating partial products from p_1. After being summed from 7 partial products to 2, a multiplexer and a 2-bit shift are used to synthetically multiply by 5. The same adder that computes the final sum for 32-bit multiplication is used as the final adder for division estimation.

The division estimation path is longer than the multiplication path, involving an extra round of Booth encoding and an extra addition, but operations are significantly narrower. However, since these operations share circuit nodes with multiplication, static timing analysis tools are likely to significantly overestimate the length of the new critical path. For example, the $\times 5$ multiplexer cuts off the slower part of the adder tree in the final calculation of $5p_1p_2$, but static timing tools cannot take advantage of this knowledge when estimating circuit delay.

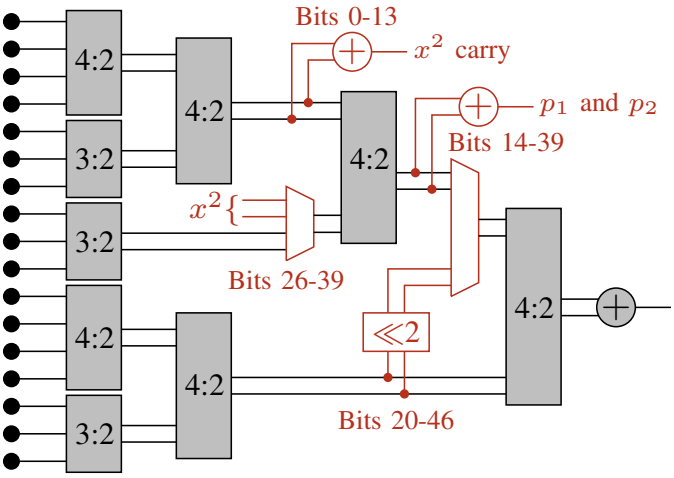

Fig. 2. Circuit diagram of compressor trees used for columns of the MUL/DIV circuit, showing added hardware in red for $1/D$ approximation present on some bits.

TABLE III
COMPARISON WITH COMMON MULTIPLIER TREES

	Column Size	Critical Path	
Architecture	**Adder Cells**	**Adder Cells**	**Layers**
Array Multiplier	15	15	15
Wallace Tree	15	6	6
4:2 Compressor Tree	15	7	5
This Work	15	8	5

A. Analysis of Circuit Size Expansion

A comparison to common multiplier tree circuits is shown in Figure 3 and Table III. Assuming a naive 4:2 compressor implementation that uses two chained full adder cells, the summation tree used is not optimal, needing 8 full adder cells on the critical path from one input to the output, instead of 6 for the optimal Wallace tree multiplier. However, with an optimized 4:2 compressor, we use the same number of logic layers as a 4:2 compressor tree. All column implementations use the same number of adder cells, and likely a similar area when optimized.

An estimate of the extra gate count required for division estimation is shown in Table IV. Cuts in the carry path show up as AND gates on the carries between columns, and cuts on the sum path are multiplexers within the summation trees on each column. The finalization of p_1 and p_2, including the production of the low carry bits of x^2, also needs extra adders. This results in an additional 40 bits of binary addition. Multiplexing within the adder trees adds an extra 92 multiplexers to the partial product array and adder tree. Multiplexing sum terms for p_1 and p_2 to construct the linear terms adds an extra bit to 106 existing multiplexers, and adding an extra input to the booth encoders and partial product generation circuits results in 70 multiplexers on the partial product generation path. The end result is a theoretical expansion of the multiplier by under 400 gates.

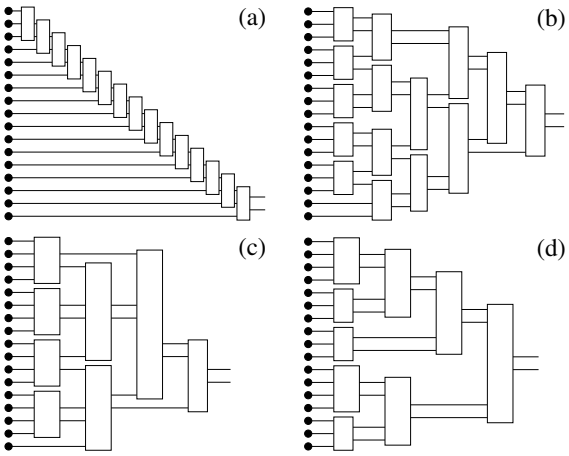

Fig. 3. Comparison of this multiplier architecture against common multiplier trees. (a) Array multiplier. (b) 3:2 compressor Wallace tree. (c) 4:2 compressor tree. (d) This work.

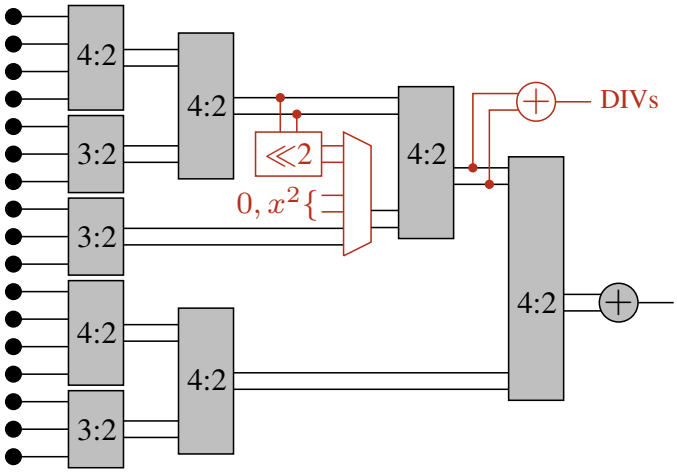

Fig. 4. Circuit diagram of the two-cycle MUL/DIV circuit. Added circuits for division, shown in red, are on bits 0–39, and compute p_1 and p_2 in one cycle, and $5p_1p_2$ in the next.

TABLE IV
ESTIMATION OF CIRCUIT ELEMENTS ADDED FOR DIVISION APPROXIMATION

	Count	Circuit Element
Generation of x^2		
Multiplexers for booth encoders	14	2:1 MUX
Multiplexers for partial products	26	2:1 MUX
Computation of carry inputs to p_1 and p_2	14	Adder Cell
Generation of p_1 and p_2		
New nonzero bits in the multiplier array	6	Adder cell
Carry chain breaks	10	AND gate
Multiplexers for adding x^2 to p_1	28	2:1 MUX
Multiplexers to override partial products	106	2:1 MUX
Final computation adders	26	Adder Cell
Final product computation		
Multiplexers for booth encoders	13	2:1 MUX
Multiplexers for partial products	24	2:1 MUX
Multiplexers for multiplication by 5	32	2:1 MUX
Output		
Output MUX port for $1/x$	32	2:1 MUX
Total Added Circuit Elements:	275	2:1 MUXes
	46	Adder Cells
	10	AND Gates

B. Approximating in Two Cycles

In systems where the multiplier takes multiple cycles or where CPU frequency is critical, the division approximation step can be split into two cycles by computing p_1 and p_2 in the first cycle, and then $5p_1p_2$ in the second cycle. A diagram of this circuit is shown in Figure 4. We move both halves of the division approximation to the top of the multiplier tree, and expand one of the multiplexers from the one-cycle variant to perform the function of both multiplexers.

This circuit moves the output port for division results to the section of the multiplier tree covering only the first 10 partial products, leaving the bottom part of the tree untouched. The output of the first cycle is a single word containing the concatenation of p_1 and p_2, and the second cycle produces the final approximation. The adders used originally for the carry from the x^2 calculation and the calculations of p_1 and p_2 are combined such that they can also be used as the final CPA for the calculation of $5p_1p_2$. Despite the movement of the adders, multiplexing zeros into the calculation leaves the results unchanged.

As a result, rather than adding 41 2:1 multiplexers to the adder tree, a total of 40 3:1 multiplexers are added, one in each of the bottom 40 bits of the multiplier. The added sum terms shown in Figure 1 are still present. Unlike in the one-cycle case, only the top 10 partial products and the top 7 Booth encoding circuits will need to have multiplexed inputs, although all of those multiplexers are widened from 2:1 to 3:1. The overall circuit size will be similar to the one-cycle division case, generally using fewer, wider multiplexers, and the same number of other extra gates.

With such a circuit, division will take an extra clock cycle, but the increase in operating frequency should more than make up for the added latency in cores used for CPU-intensive applications. This circuit also avoids re-entering the Booth encoding circuits on the multiplier array, so pessimism on static timing analysis is minimized.

IV. Division State Machine

To examine the full cost of hardware for fast division using this circuit, we designed a state machine that runs Newton-Raphson division using the augmented multiplier and common components of microcontroller ALUs. The state machine is intended to be integrated into the decode and execute stages of a microcontroller, but the standalone state machine allows us to check the correctness of the algorithm and understand the area cost.

The division state machine uses hardware shown in Figure 5. The state machine makes use of the augmented multiplier, a shifter, the addition components from a microcontroller

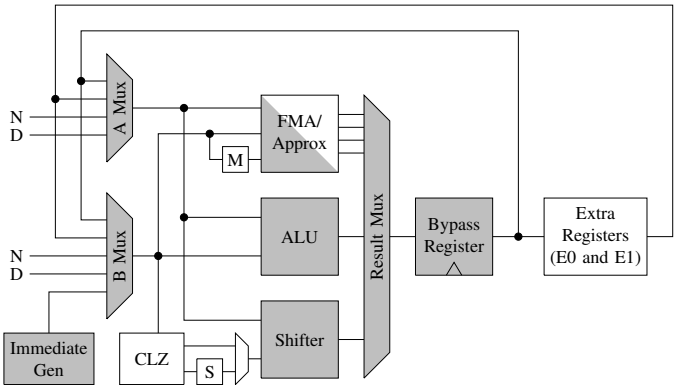

Fig. 5. Diagram of the datapath used by the division state machine. Common microcontroller components used for other instructions are in gray.

ALU, and a leading zero counter used to left-justify the divisor. Small microcontrollers don't always include leading zero counters, but the circuit is comparatively small [17]. We also use two extra registers, E0 and E1, that are assumed to be added to the register file.

The Bypass and extra in the division state machine are intended to simulate a 3-stage microcontroller with a 2-cycle write-to-read latency, a bypass register covering that latency, and two extra registers in the register file. One extra register is often free in some microcontrollers that have a synthetic zero register (*eg* RISC-V) [1]. This write-to-read latency corresponds to a core with separate execute and write-back stages, or a core with separate decode and execute stages whose register values are loaded in the decode stage.

We also add a few extra capabilities to the multiplier:

- Adding 2^{32} to the Booth-encoded operand on a signed multiplication
- Negating a multiplication
- Right shifting the output by 1

All of these come at the cost of relatively few gates: the first extra capability comes from enabling the most significant partial product, which is not normally used for signed multiplication. Negating a signed multiplication is enabled by XORing all of the NEG bits of the Booth code, negating each partial product before they are summed. Right shifting the output comes from adding an extra port to the result multiplexer: going from 3 multiplier result terms to 4.

Finally, FMA capability was added to the bottom 32 bits of the multiplier using a shadow register to avoid adding a third operand port. This allows us to do remainder calculation in one cycle, but is a comparatively expensive change, as it requires an expansion from a 17-term adder tree to an 18-term adder tree. Alternatively, two-cycle remainder calculation can be done using multiplication and subtraction, which would add one cycle to every non-trivial division. The FMA functionality is not used for any other state machine step.

A. The State Machine Algorithm

State machine cycles are shown in Table V. The state machine begins by using the leading zero counter and the shifter to left-justify the denominator. A 5-bit register stores the leading zero count for use later.

In the second cycle, we run division estimation to produce an initial guess of the reciprocal.

$$x_0 = A_{1/D}(D)$$

To keep $D * x_0$ between 0 and 1, we underestimate in this step, using the constants from the bottom row of Table II. This allows us to only store the fractional part of x_n and Dx_n.

The state machine then uses a standard Newton-Raphson iteration rule [14]:

$$x_{n+1} = x_n(2 - Dx_n)$$

and uses the augmented multiplier to compute this update rule in two cycles. In the first cycle, we compute Dx_n and save bits [62:31] to get a result in UQ0.32 fixed point. Since we are working with strict underestimates for x_n, we know that $Dx_n < 1$. In the next cycle, to compute $x_n(2 - Dx_n)$, we treat (Dx_n) as signed since its leading digit is known to be 1, negate it, and add 2^{32}. This is isomorphic to adding 2 to the fixed point $-Dx_n$.

After a precise estimate for $1/D$ is calculated, we multiply x_2 by N. We know that x_2 is an underestimate within 1 ULP, so we have either a correct quotient at this stage or we are under by 1. We then shift and calculate an interim remainder with an FMA operation.

After this, we check whether the interim remainder is over D or under it. If it is over D, we correct the remainder and add an 11th cycle to the state machine to correct the quotient. This means that unlike most division algorithms, which compute a quotient before a remainder, this algorithm computes a correct remainder in 10 cycles, while a quotient may take 11 cycles to compute.

By accepting that our calculation may be off by 1, we avoid having to expand the datapath to recover precision losses to truncation, and we avoid any need for rounding circuits. However, expanding the datapath by adding 1 bit specifically for division should allow the elimination of the correction steps, causing the divider to terminate in 9 cycles at worst.

For signed division, most steps are the same as unsigned division. However, we also need to add conditional negation in the first and last steps of the quotient calculation, saving the sign bit along with the shift amount. With the division estimation function and error bounds we have, logical negation is sufficient, allowing the re-use of the XOR gates that may already be present on operand B for operations like subtraction. Steps 7 and 9–11 then use signed multiplication rather than unsigned.

B. Early Termination

The state machine is designed to support early termination for trivial cases and cases with a known upper bound on the position of the output's MSB, computed based on the leading zero count of N and D:

$$\text{MSB} \leq \text{CLZ}(D) - \text{CLZ}(N)$$

TABLE V
STATE MACHINE STEPS TO COMPUTE QUOTIENT AND REMAINDER

			Inputs		Operation		Output	
Cycle	**Operation**	**Equation Form**	**Operand A**	**Operand B**	**ALU/Shift**	**Multiplier**	**Source**	**Register**
1	Left-justify D	$D' = D \ll S$	D	D	$\mathtt{A} \ll (\mathtt{S} = \mathtt{CLZ(B)})$		Shift	E0
2	Estimate $1/D$	$X = A_{1/D}(D')$	BYP			$\mathtt{APPROX}_{1/\mathtt{D}}(\mathtt{A})$	Mul	E1
3	Get error (DX_0)	$T = D'X/2$	BYP	E0		$(\mathtt{A_U} \times \mathtt{B_U}) \gg 1$	Mul	
4	Refine $X_0 \to X_1$	$X = X(2 - T)$	BYP	E1		$(2^{32} - \mathtt{A_S}) \times \mathtt{B_U}$	Mul	E1
5	Get error (DX_1)	$T = D'X/2$	BYP	E0		$(\mathtt{A_U} \times \mathtt{B_U}) \gg 1$	Mul	
6	Refine $X_1 \to X_2$	$X = X(2 - T)$	BYP	E1		$(2^{32} - \mathtt{A_S}) \times \mathtt{B_U}$	Mul	
7	Calculate quotient	$Q' = NX$	BYP	N		$\mathtt{A_U} \times \mathtt{B_U}$	Mul	
8	Finish quotient	$Q = Q' \gg S$	BYP	N	$\mathtt{A} \gg (32 - \mathtt{S})$	$\mathtt{B} \to \mathtt{M}$	Shift	Q
9	Calculate remainder	$R = N - DQ$	BYP	D		$\mathtt{M} - \mathtt{A_U} \times \mathtt{B_U}$	Mul	R
10	Check remainder	$R = R \mod D$	BYP	D	$(\mathtt{R} \geq \mathtt{D})\ ?\ \mathtt{R} - \mathtt{D}$		ALU	R
11*	Fix quotient	$Q = Q + 1$	Q		$\mathtt{Q} + 1$		ALU	Q

TABLE VI
EXECUTION LATENCY CASES FOR DIVIDER STATE MACHINE

Case	**Condition**	**Total Cycles**
Divide by zero	$S = 32$	3
Power of 2	$(D \ll S) = 0.5$	3
Known 0 result	MSB < 0	4
8-bit precision	$0 \leq$ MSB < 8	6–7
16-bit precision	$8 \leq$ MSB < 16	8–9
Full precision	$16 \leq$ MSB	10–11

TABLE VII
FPGA SYNTHESIS COMPARISON OF CIRCUITS

Synthesized Circuit	**Logic Elements**	**Synthesized f_{max}**
Wallace Tree Multiplier	2,172	80 MHz
Multiplier Tree Only	2,175	76 MHz
Division Estimation Only	1,310	82 MHz
Multiply/Divide (One Cycle)	2,340	43 MHz
Multiply/Divide (Two Cycle)	2,345	71 MHz

This can be computed in the second state machine cycle, during the estimation stage, when both the ALU and the operand B port are otherwise idle.

Early termination conditions are shown in Table VI. For trivial cases, we can skip division estimation and only do steps 1 and 8. For cases with an 8-bit precision requirement, we skip steps 3–6. For 16-bit precision, we skip steps 5–6.

To check these conditions, early termination involves extra hardware to check for power of two divisions and compute the position of the MSB, as well as adding three extra states to the state machine to cover the trivial cases of division by zero or a power of two, and known zero results. To detect powers of two, the early termination logic needs extra hardware to check whether the left-justified divisor was equal to 0.5 (0x80000000), adding an extra circuit smaller than a leading zero counter. A system that would rather have constant 11-cycle division can eschew these circuits for a small area savings.

V. EVALUATION

A. Area and Frequency of Initial Approximation Circuit

The augmented multiplier was synthesized and run on a Max 10 FPGA. The Max 10 was selected because its logic fabric uses 4-input lookup tables, which makes it a good proxy for ASIC synthesis of this circuit.

Synthesis results are summarized in Table VII, including counts of logic elements (4-input LUTs), and an estimation of maximum operating frequency from the Quartus Static Timing Analysis program. A standard Wallace tree multiplier using 3:2 compressors synthesizes in the same number of logic cells as a multiplier using the hybrid tree architecture from Figure 2. Both synthesize to similar maximum frequencies, 80 MHz for the Wallace tree and 76 MHz for our hybrid tree. When synthesized alone, the division estimation circuits use 1,310 logic elements, about half the size of the multiplier, and synthesize to a maximum frequency of 82 MHz.

When the division approximation circuits are added, the combined multiplier expands from 2,175 logic elements to 2,340 logic elements, adding 8% more LEs to support division. When the division circuits are added, the second path through the multiplier significantly reduces the estimated maximum frequency of the multiplier from 76 MHz to 43 MHz, a 43% reduction in frequency. However, the large drop in f_{max} is at least partly explained by pessimism in static timing analysis.

The two-cycle combined multiply/divide circuit synthesizes to a similar size as the one-cycle circuit, but reaches an analyzed f_{max} of 71 MHz, recovering the pessimism but reflecting the delay of added logic gates and multiplexers in the critical path of the multiplier.

For comparison with other systems that use iterative division, [5] uses a lookup table of 69 Kbits to cover both reciprocal and floating-point square root calculations. [6] uses a 1.3 Kbit lookup table for initial reciprocal approximation, but requires 3 iterative refinement rounds to reach 24-bit precision, so it may not have been capable of 32-bit precision without an extra refinement round.

TABLE VIII
FPGA SYNTHESIS RESULTS OF THE DIVISION STATE MACHINE

Component	Logic Elements
Division State Machine With Early Termination	**3,260**
Multiply/Divide Components	**2,673**
FMA with Division Estimation	2,491
Leading Zero Count	47
Power of 2 Detection	13
Control State Machine	56
Extra Registers	66
Common Components	**587**
Bypassing, Adder, and Datapath Multiplexers	357
Funnel Shifter	230
Comparison Baseline—Basic Datapath	**2,759**
Booth + Wallace Tree Multiplier	2,172
Common Components	587
Added Fast Division Logic vs. Basic Datapath	**501**

B. Analysis of Division State Machine

The state machine was also synthesized and run on a Max 10 FPGA to evaluate its size and performance. The synthesized state machine with early termination takes 3,260 logic cells, and runs at 42 MHz. A breakdown of the size of circuit elements in the state machine is in Table VIII. The multiplication and division components consume most of the area of the circuit, but the vast majority of these components come from the multiplier itself. The components responsible for division account for 501 total additional logic cells when compared against a baseline of a multiplier, a shifter, an ALU, and a basic datapath, representing an extra 18% area in the datapath to support fast division.

If the baseline machine also includes two extra registers and some common bit manipulation operations, like popcount (which handles power of 2 detection) and leading zero count, the added cost for fast division is only 375 logic elements, or a 14% increase in area.

VI. CONCLUSIONS

A microarchitecture for 32-bit integer division in small devices using Newton-Raphson iteration has been presented. This system uses a quantized fourth-order Remez approximation of $1/x$ instead of a lookup table, re-using the hardware in the integer multiplier to compute the polynomial in either one or two cycles depending on frequency requirements, at a cost of an 8% area increase in the multiplier. Further, we show a state machine that uses this approximation circuit to compute 32-bit integer divisions 3–4$\times$ faster than a radix-2 division unit with a similar hardware footprint. The division state machine completes remainder operations in 3–10 cycles, and quotients in 2–11 cycles, with arbitrary 8-bit operations terminating in 6–7 cycles and 16-bit operations terminating in 8–9 cycles.

When synthesized in an FPGA model, the division state machine increases the area of a microcontroller datapath by 18% compared to a datapath with no division. The comparative performance increase available for a modest increase in area makes this circuit a viable alternative to methods like radix-2 non-restoring division in area-constrained applications.

ACKNOWLEDGEMENTS

The author would like to thank the anonymous reviewers for their comments and suggestions.

REFERENCES

[1] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*, RISC-V Foundation, 2017.

[2] *MicroBlaze Processor Reference Guide (UG984)*, Online, Xilinx Inc., 2021. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug984-vivado-microblaze-ref.pdf

[3] E. Matthews and L. Shannon, "TAIGA: A new RISC-V soft-processor framework enabling high performance cpu architectural features," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[4] J. D. Bruguera, "Low-latency and high-bandwidth pipelined radix-64 division and square root unit," in *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, 2022, pp. 10–17.

[5] S. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7™ microprocessor," in *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336)*, 1999, pp. 106–115.

[6] A. Naini, A. Dhablania, W. James, and D. Das Sarma, "1 GHz HAL SPARC64® dual floating point unit with RAS features," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, 2001, pp. 173–183.

[7] A. Fog, "Instruction tables," Online, 2022. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[8] D. Das Sarma and D. Matula, "Faithful bipartite rom reciprocal tables," in *Proceedings of the 12th Symposium on Computer Arithmetic*, 1995, pp. 17–28.

[9] K. Jun and E. E. Swartzlander, "Modified non-restoring division algorithm with improved delay profile and error correction," in *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, 2012, pp. 1460–1464.

[10] E. Matthews, A. Lu, Z. Fang, and L. Shannon, "Rethinking integer divider design for fpga-based soft-processors," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 289–297.

[11] L. Li, M. Gautschi, and L. Benini, "Approximate DIV and SQRT instructions for the RISC-V ISA: An efficiency vs. accuracy analysis," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[12] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[13] D. Lemire, O. Kaser, and N. Kurz, "Faster remainder by direct computation: Applications to compilers and software libraries," *Software: Practice and Experience*, vol. 49, no. 6, pp. 953–970, feb 2019.

[14] M. Flynn, "On division by functional iteration," *IEEE Transactions on Computers*, vol. C-19, no. 8, pp. 702–706, 1970.

[15] W. Fraser, "A survey of methods of computing minimax and near-minimax polynomial approximations for functions of a single independent variable," *J. ACM*, vol. 12, no. 3, p. 295–314, jul 1965.

[16] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–8.

[17] J. Miao and S. Li, "A design for high speed leading-zero counter," in *2017 IEEE International Symposium on Consumer Electronics (ISCE)*, 2017, pp. 22–23.